LAB1 : DEEP LEARNING

Neural Network : First implementation with a linear model

[Florian Bertelli][1], [Timothée Babinet][2], [Ramine Hamidi][3]

ABSTRACT

In the following document our aim is to implement a neural network, using numpy, without using a neural network library (like TF, Pytorch). The task will be to predict the digits, with the famous MNIST database.

Data description

- The data are downloaded from the MNIST database, which contains 50 000 images of hand-written digits, each image being associated with the corresponding label (integer). Each image has a resolution of 28 * 28 pixels.

Summary

- This lab contains a full implementation of a neural network, with the training of a linear classifier using : $o = \text{softmax}(Wx + b)$
  * x is an input image that is represented as a vector, each value being the grey intensity of a pixel
  * W and b are the parameters of the classifier
  * softmax transforms the output weight (logits) into probabilities
  * o is a column vector that contains the probability of each category

- Our loss function, that we will try to minimize during the training is :

$$\mathcal{L}(x, gold) = -\log \frac{\exp(x[gold])}{\sum_j \exp(x[j])}$$

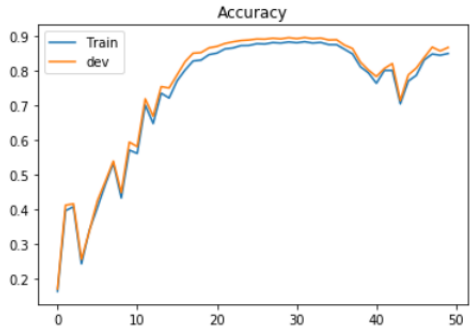
Parameters initialisation

- As we are doing an affine transform we decided to initialise
  - The bias $b_0 = 0$
  - The weights $W \sim [-\frac{\sqrt{6}}{\sqrt{n_j+n_{j+1}}}; \frac{\sqrt{6}}{\sqrt{n_j+n_{j+1}}}]$

Naive Implementation Vs Optimized Implementation

- First, we did a naive implementation. Indeed, we just implemented a neural network, where the learning rate is fixed before and not adaptive, which keeps the last W and b (which are not necessarily the best).
  With our best implementation, and 50 epochs, we get:
  1. Accuracy on train data : 0.87872
  2. Accuracy on dev data : 0.8939
  3. Accuracy on test data : 0.8888

  To obtain such results, we first had to get a good learning rate. Indeed, the first one, 0.01 was too big, and we didn't manage to reach something close enough to the global optimum. Indeed, with a learning rate equal to 0.01, we got an accuracy on test data of 0.8516. With a learning rate equal 0.01 we have the following accuracy for each epoch:



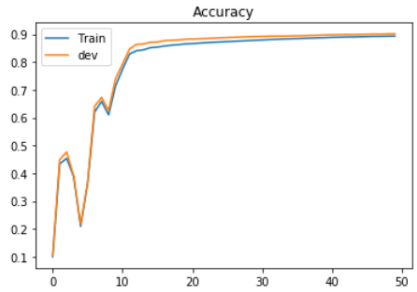
  We can see that sometimes the accuracy decreases. This means that the learning rate is too big, and we overtake the optimum.

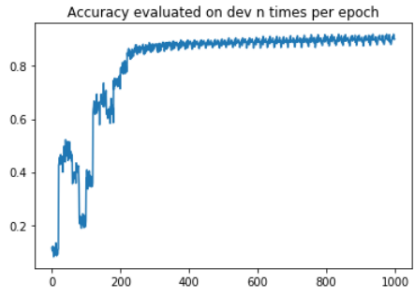
- Then, we tried a smarter implementation of the neural network. Below is explained how we did it, and why:
  1. Instead of evaluating on dev dataset after each loop on the training data, you can also evaluate on dev n times per epoch. It allows us to have more results and to observe the variation of the results inside the dev dataset. The more the variation is important, the more likely is our model to overfit, and this is something we want to avoid.
  2. Shuffle the data before each epoch : this allows us to escape a global minimum and to reach the global minimum. You want to shuffle your data after each epoch because you will always have the risk to create batches that are not representative of the overall dataset, and therefore, your estimation of the gradient will be off.
  3. Instead of memorizing the parameters of the last epoch only, you should have a copy of the parameters that produced the best value on dev data during training, and evaluate on test with those parameters instead of the parameters you got after the last epoch. The reason here is quite simple: we want to keep the best parameters, and the last ones aren't always the best.

4. Learning rate decay: if you do not observe improvement on dev, you can try to reduce the step size. Reducing the learning rate allows us to start with big steps, and so to avoid being trapped in a local minimum. Then, with the learning rate decreasing, it gives us enough precision to get as close to the global minimum as we can.

In this part, we've got the following results with 50 epochs and an initial learning rate of 0.001, and a decreasing factor of 0.8 for the learning rate :

1. Accuracy on test data : 0.8992
2. Accuracy on train data : 0.8929
3. Accuracy on dev data : 0.9011

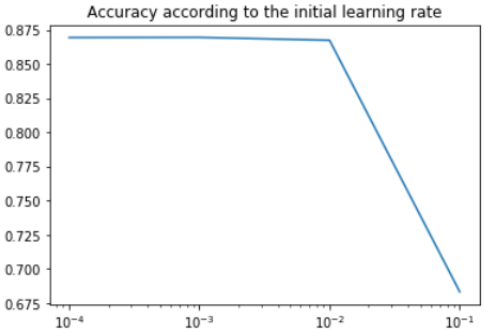




On these two graphics, we can see that the accuracy is increasing with the number of epochs. There are some decreases at the beginning because the learning rate was too big and so sometimes we went too far from the optimum. Furthermore, we can see that if we compute N times the accuracy on the dev dataset, there are variations. They aren't too big, but it reminds us why shuffling is important to avoid to learn on bad batches.

Learning rate initialisation

- To determine the initial learning rate that performs better, we run the training algorithm for 10 epochs for different learning rates : [0.1 , 0.01 , 0.001 , 0.0001], and we compare the results.



According to this result we choose an initial learning rate equal to 0.001, because it gives us the best results for 10 epochs.

Conclusion

To conclude, the optimized version of the naive neural network gave us the best results. Indeed, it allows the network to learn more (thanks to the adaptive learning rate) and to avoid overfitting (using data shuffling). This leads to better performances. However, even if the "gain" isn't really big, for more complicated tasks where the neural network architecture is really sophisticated, all these improvements will be really important to train efficiently.